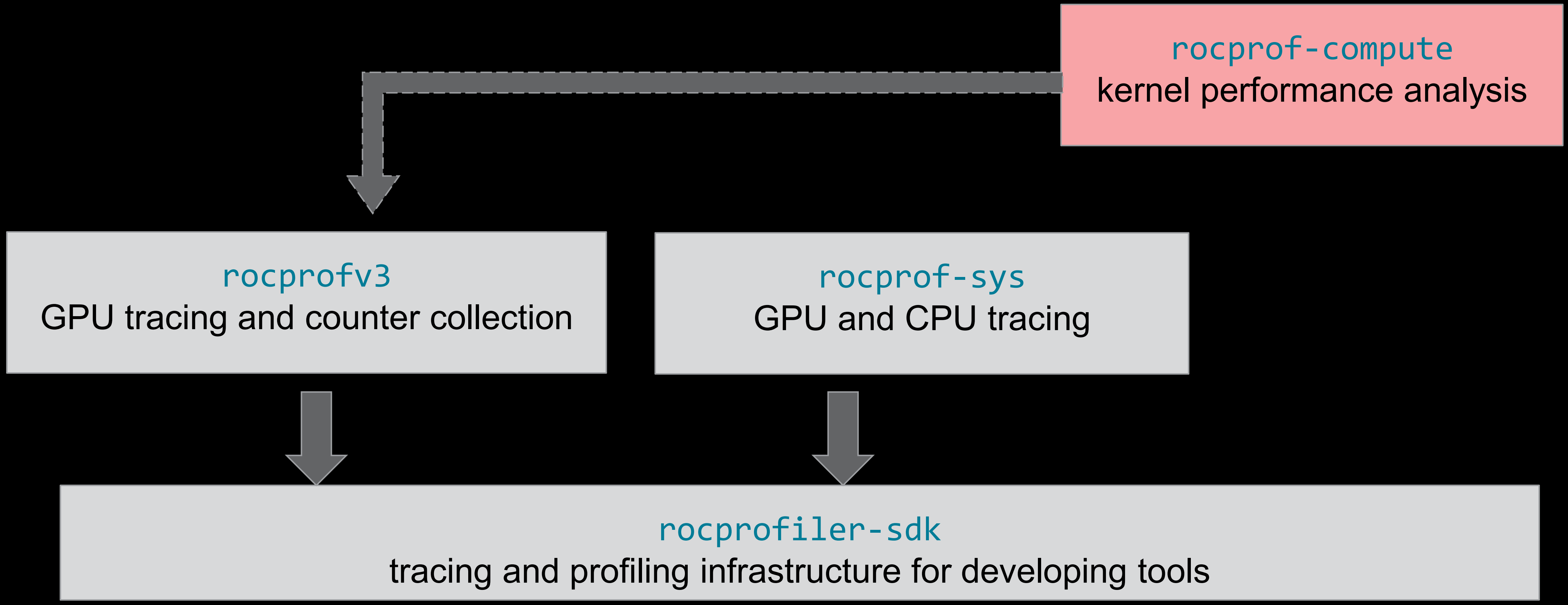
Introduction to rocprof-compute

Presenter: Bob Robey
AMD @ Tsukuba University
Oct 21-23, 2025

AMD
together we advance_

AMD has three GPU profiling tools



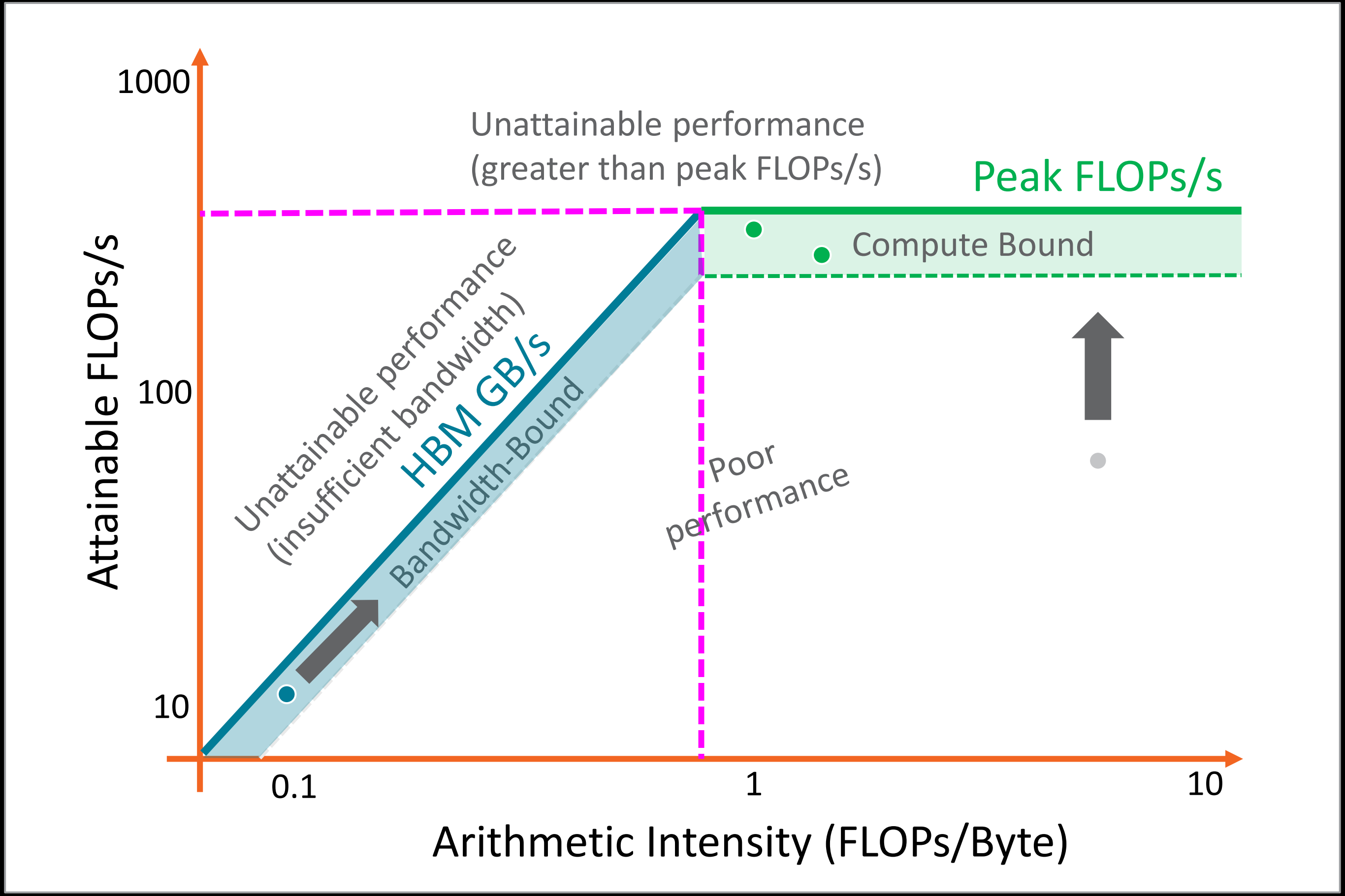
What is rocprof-compute?

- rocprof-compute is a GPU kernel performance analysis tool added to ROCm with version 6.3
  - Previously (before ROCm 6.3) called Omniperf

- Most notable features:
  - Roofline analysis to quantify performance of GPU kernels based on achievable hardware limits
  - Kernel comparison to quantify code changes and confirm their impact on hardware
  - Derived performance metrics that provide deep insight into kernel performance
  - Support for speed of light and memory chart

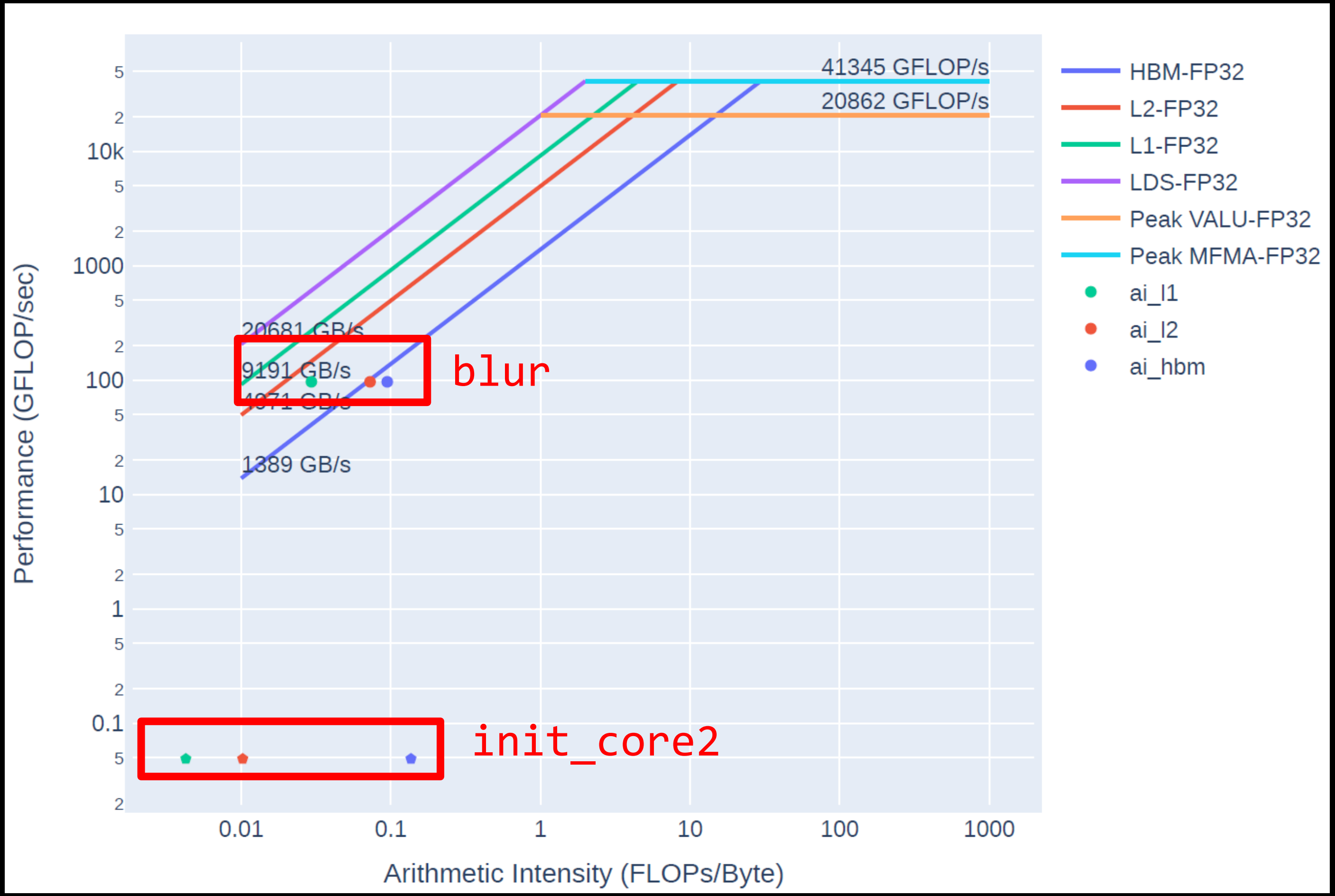
Background – What is roofline?

- Attainable FLOPs/s =
  - $min\begin{cases} Peak\ FLOPs/s \\ AI * Peak\ GB/s \end{cases}$
- Machine balance:
  - Where $AI = \frac{Peak\ FLOPs/s}{Peak\ GB/s}$
- Five performance regions:
  - Unattainable compute
  - Unattainable bandwidth
  - Compute bound
  - Bandwidth bound
  - Poor performance



Visualize rooflines with rocprof-compute

```
rocprof-compute profile -n rooflines_PDF --roof-only --kernel-names -- ./GhostExchange -x 1 -y 1 -i 20000 -j 20000 -h 2 -t -c -I 100
```



Note: In some rocprof-compute versions FP32 and FP64 lines overlapping – this is fixed in later versions.

Get kernels info with rocprof-compute

- To generate profiling data run:

```
rocprof-compute profile -n v1 --no-roof -- ./GhostExchange -x 1 -y 1 -i 200 -j 200 -h 2 -t -c -I 100
```

- You can then display the IDs of the kernels involved by doing:

```
rocprof-compute analyze --list-stats -p workloads/v1/MI300A_A1/
```

blur kernel has ID 0

Detected Kernels (sorted decending by duration)

	Kernel_Name
0	blur(double**, double**, int, int) [clone .kd]
1	init_core(double**, int, int, int, double) [clone .kd]
2	enforce_bcs_left(double**, int, int) [clone .kd]
3	enforce_bcs_rght(double**, int, int, int) [clone .kd]
4	enforce_bcs_top(double**, int, int, int) [clone .kd]
5	enforce_bcs_bot(double**, int, int) [clone .kd]
6	init_core2(double**, int, int, int, int, int, int, int, int, int) [clone .kd]
7	__amd_rocclr_initHeap.kd

Get kernel dispatch ID with rocprof-compute

- The command below will also show you the IDs for each kernel dispatch. Let's consider `blur`:

```
rocprof-compute analyze --list-stats -p workloads/v1/MI300A_A1/ | grep blur
```

```
  0 | blur(double**, double**, int, int) [clone .kd]                  |
  7 |        7 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 12 |       12 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 17 |       17 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 22 |       22 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 27 |       27 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 32 |       32 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 37 |       37 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 42 |       42 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 47 |       47 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 52 |       52 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 57 |       57 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 62 |       62 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 67 |       67 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 72 |       72 | blur(double**, double**, int, int) [clone .kd]        |  2 |
 77 |       77 | blur(double**, double**, int, int) [clone .kd]        |  2 |
```

The dispatch IDs correspond to the launch order of the kernels during the run

Compare different kernel implementations

- Modify the `blur` and `init_core` kernel grid size and block size at `GhostExchange.hip:207` from:

```
dim3 grid((isize+63)/64, (jsize+3)/4, 1);          dim3 block(64, 4, 1);
```

- To:

```
dim3 grid((isize+255)/256, (jsize+3)/4, 1);        dim3 block(256, 4, 1);
```

- Then compile a generate the profiling data for this new version:

```
rocprof-compute profile -n v2 --no-roof -- ./GhostExchange -x 1 -y 1 -i 200 -j 200 -h 2 -t -c -I 100
```

- You can compare the two versions by using `rocprof-compute analyze`:

```
rocprof-compute analyze -p workloads/v1/MI300A_A1 -p workloads/v2/MI300A_A1 --block 16.2 17.2
```

17. L2 Cache
17.2 L2 - Fabric Transactions

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
17.2.9	Read Latency	356.34	363.41 (1.98%)	-1210.06	188.86	160.68 (-14.92%)	1897.06	687.0 (-63.79%)	Cycles

Not specifying any dispatch in the `rocprof-compute analyze` command above will show averaged values

Guided exercises

1. Launch parameters
2. LDS occupancy limiter
3. VGPR occupancy limiter
4. Strided data access pattern / representative problem size

Guided exercises: Logistics/Preamble

```
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/rocprof-compute
```

- Feel free to clone the above repo and start working through the exercises
  - The READMEs are comprehensive walkthroughs on their own, I'll provide highlights in the talk
  - The numbers shown in the READMEs were generated using MI210 and MI300A accelerators, and the accelerator used is made clear in each case

- To generate the output for these slides use rocprof-compute from ROCm 6.4.0
  - This is a module available to you on the training environment: `module load rocprofiler-compute/6.4.0`

- WARNING: For educational purposes implementations in these exercises are not fully-optimized kernels

Guided exercises: Representative optimization tasks

- The exercises are roughly in order of ease of development effort and performance impact:
  - Exercise 1: Verify reasonable launch parameters
  - Exercise 2: Attempt to cache data in shared memory
  - Exercise 3: Determining a source of unexpected resource usage
  - Exercise 4: Verifying efficient data access patterns and representative problem sizes
- Though we use a simple HIP code, we have verified that rocprof-compute works with Fortran + OpenMP® codes, and the material on these slides should apply to those codes as well
- The underlying code kept simple (and barely mentioned) to emphasize the optimization techniques
- These slides are intended as a “Cheat Sheet” starting point providing:
  - Commands to filter through output for common optimization concerns
  - Some optimization direction given certain output

Guided exercises: Optimizing yAx kernel

- We'll be looking at a relatively simple kernel that solves the same problem in each exercise
  - yAx is a vector-matrix-vector product that can be implemented in serial as:

```
double result = 0.0;
for (int i = 0; i < n; i++){
  double temp = 0.0;
  for (int j = 0; j < m; j++){
    temp += A[i*m + j] * x[j];
  }
  result += y[i] * temp;
}
```

- Where:
  - A is a 1-D array of size n*m
  - x is an array of size m
  - y is an array of size n

Exercise 1: First things first, generate a roofline

- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `rocprof-compute profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/MI300_A1` folder.
    - `--roof-only` generates PDF roofline plots, and does **not** generate any non-roofline profiling data
    - `--kernel-names` generates a separate PDF showing which kernel names correspond to which icons in the roofline

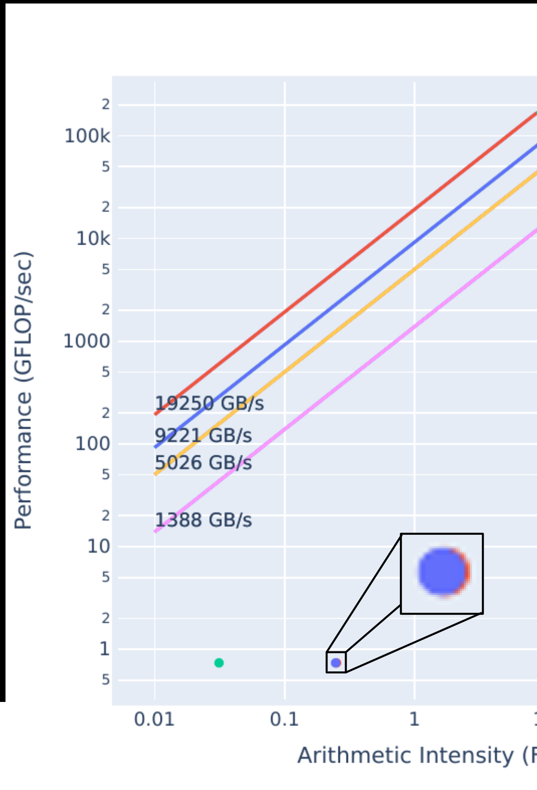
- Rooflines are a useful tool in determining which kernels are good optimization targets
  - Only one perspective of performance, kernel runtime cannot be inferred from the roofline

- Generated PDF roofline plots can have overlapping data points but should still be instructive
  - There are fixes to this, but they may be difficult to setup for different cluster installations
  - Generating the PDF plots from the command line interface should always work

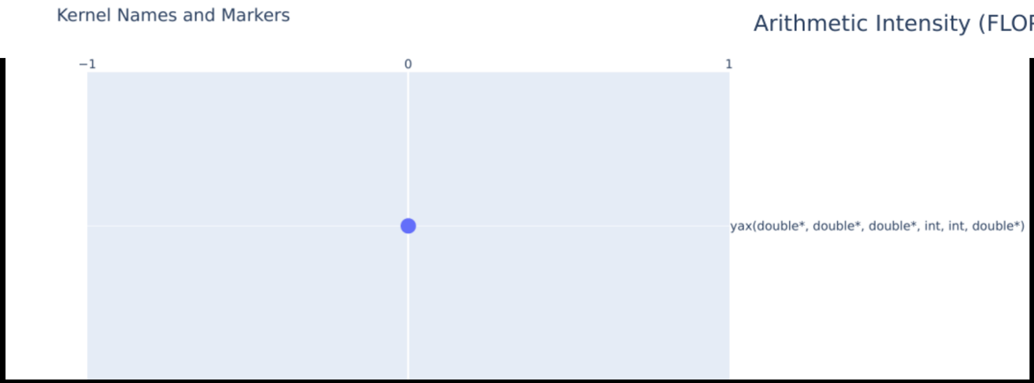
- Complete sets of roofline plots and commands can be found in the READMEs for each exercise

[Public]

Exercise 1: Roofline plots



Note: The L2 data point is hidden behind the HBM data point



Kernel legend in a separate PDF

Oct 21-23, 2025

AMD together we advance_

Exercise 1: Prep to find kernel launch parameters

- Launch parameters are given at the time of the kernel launch, as in lines 49 and 54:
  - `yax<<<grid,block>>>(y,A,x,n,m,result);`
    - Where grid and block are the kernel yax's launch parameters
  - In problem, `grid = (4,1,1)`, and `block = (64,1,1)`
  - In solution, `grid = (2048,1,1)`, and `block = (64,1,1)`
- Sometimes launch parameters can be obfuscated by OpenMP® and other parallelism layers
- rocprof-compute can easily show launch parameter information regardless of the code
  - You just need the dispatch ID – other forms of filtering may report aggregate launch parameters
- To generate profiling data, use the commands:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`
    - `--no-roof` saves time by not generating roofline data – profile commands can take a while
- Real benchmarks can take prohibitively long – use smaller representative problems when possible

[Public]

Exercise 1: CLI rocprof-compute comparisons are easy

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

Using problem as the baseline, and solution as the comparative

```
INFO Analysis mode = cli
INFO [analysis] deriving rocprofiler-compute metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	543201153.00	9589864.0 (-98.23%)	543201153.00	9589864.0 (-98.23%)	543201153.00	9589864.0 (-98.23%)	100.00	100.0 (0.0%

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

56.6x speedup

Typically, difficult to pre-determine optimal launch parameters, so some experimentation is often necessary

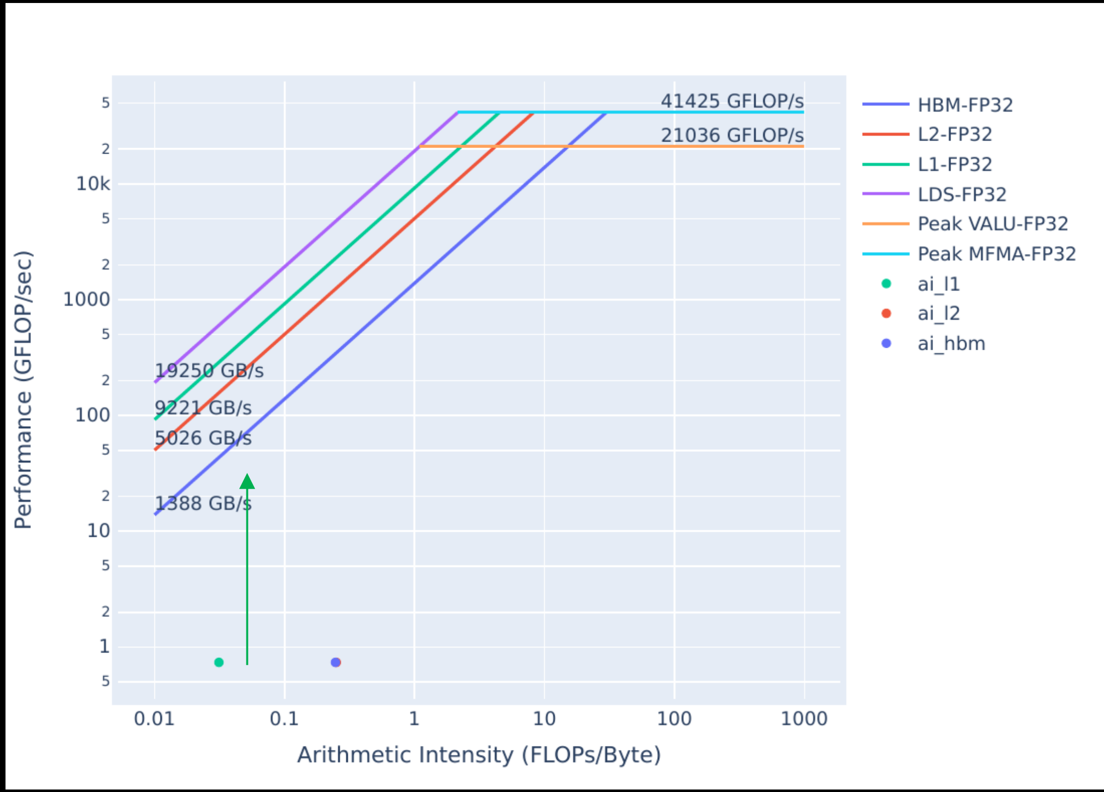
```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
7.1.0	Grid Size	256.00	131072.0 (51100.0%)	130816.00	256.00	131072.0 (51100.0%)	256.00	131072.0 (51100.0%)	Work items
7.1.1	Workgroup Size	64.00	64.0 (0.0%)	0.00	64.00	64.0 (0.0%)	64.00	64.0 (0.0%)	Work items
7.1.2	Total Wavefronts	4.00	2048.0 (51100.0%)	2044.00	4.00	2048.0 (51100.0%)	4.00	2048.0 (51100.0%)	Wavefronts

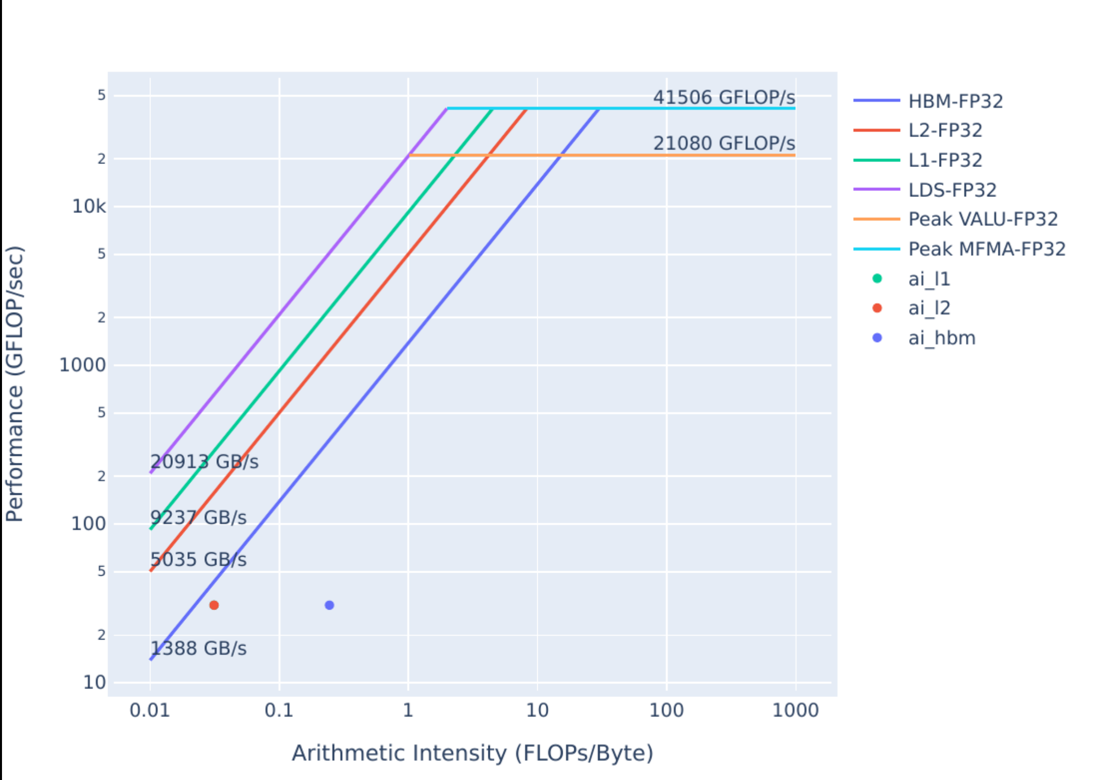
Increased launched wavefronts, which increases grid size

Exercise 1: Comparing problem and solution roofline plots

Problem FP32 Roofline Plot



Solution FP32 Roofline Plot



Generally, moving up and to the right is good

Exercise 1: It’s easy to check launch parameters

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

- `--block` filters the output to only show launch parameters

- Good launch parameters essential to a performant GPU kernel
  - Determining which parameters give the best performance usually requires experimenting

- Can be difficult to track down where launch parameters are set in code (OpenMP® may decide)

Exercise 2: Diagnosing shared memory occupancy limiter

- Using LDS (Local Data Store) shared memory to cache re-used data can be effective optimization strategy

- Using too much LDS can restrict occupancy however, and reduce performance

- LDS allocation example:
  - `__shared__ double tmp[fully_allocate_lds];`

- Two solutions proposed in the exercises:
  - `solution-no-lds` removes the LDS allocation, and thus the occupancy limiter
  - `solution` reduces the size of the LDS allocation, removes occupancy limiter, and is faster than `solution-no-lds`
    - This is the solution used to generate the rocprof-compute output in the next slide

- rocprof-compute makes it easy to determine if LDS allocations restrict occupancy
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`

Exercise 2: LDS occupancy limiter – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.7

INFO Analysis mode = cli
INFO [analysis] deriving rocprofiler-compute metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	7225180.00	5736816.0 (-20.6%)	7225180.00	5736816.0 (-20.6%)	7225180.00	5736816.0 (-20.6%)	100.00	100.0 (0.0%)

1.26x speedup

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

```
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Peak	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	175.66	418.68 (138.35%)	3.33	Wavefronts	7296.00	7296.0 (0.0%)	2.41	5.74 (138.31%)

+ ~3% Occupancy (overall)

```
--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.7	Insufficient CU LDS	57.33	0.0 (-100.0%)	-57.33	57.33	0.0 (-100.0%)	57.33	0.0 (-100.0%)	Pct

Sharp decrease in Workgroup Manager stat

Exercise 2: Use SPI stats to determine if LDS limits occupancy

- Occupancy limiters can negatively impact performance
  - Occupancy increases don't always correspond to increased performance

- Workgroup Manager (SPI – Shader Processor Input) stats in rocprof-compute indicate whether a kernel resource limits occupancy

- You can get the Workgroup Manager stat for LDS for a single kernel with dispatch ID 1:
  - `rocprof-compute analyze -p workloads/problem/MI300_A1 --dispatch 1 --block 2.1.15 6.2.7`

Note:

- In rocprof-compute, the Workgroup Manager "insufficient resource" stats are percentages, meaning:
  - The magnitude of these fields **does not** necessarily indicate how severely occupancy is impacted
    - Changes to the Workgroup Manager stat do not directly translate to changes to overall occupancy, necessarily
  - If two fields are nonzero, the larger number indicates that resource is limiting occupancy more

Exercise 3: Diagnosing a register occupancy limiter

- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - The solution simply removes the assert
  - Admittedly the occupancy limit is very minor, but this is a good excuse to look at register usage.

- The types of registers on AMD GPUs are:
  - VGPRs (Vector General Purpose Registers): registers that can hold distinct values for each thread in the wavefront
  - SGPRs (Scalar General Purpose Registers): uniform across a wavefront. If possible, using these is preferable
  - AGPRs (Accumulation vector General Purpose Registers): special-purpose registers for MFMA (Matrix Fused Multiply-Add) operations, or low-cost register spills

- Using too many of one of these register types can impact occupancy and negatively impact performance

- We use the same profile commands to get the profiling data:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`

Exercise 3: Register occupancy limiter – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9993665.00	9666265.0 (-3.28%)	9993665.00	9666265.0 (-3.28%)	9993665.00	9666265.0 (-3.28%)	100.00	100.0 (0.0%

Minor speedup

```
0.2 Dispatch List
<omitted>
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Pea	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	430.98	427.36 (-0.84%)	-0.05	Wavefronts	7296.00	7296.0 (0.0%)	5.91	5.86 (-0.85%)

Similar occupancies

```
--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.5	Insufficient SIMD VGPRs	0.06	0.0 (-99.7%)	-0.06	0.06	0.0 (-99.7%)	0.06	0.0 (-99.7%)	Pct

Minor change in Workgroup Manager stat

Exact values might be slightly different, but conclusion stay the same

```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
7.1.5	VGPRs	92.00	32.0 (-65.22%)	-60.00	92.00	32.0 (-65.22%)	92.00	32.0 (-65.22%)	Registers
7.1.6	AGPRs	132.00	0.0 (-100.0%)	-132.00	132.00	0.0 (-100.0%)	132.00	0.0 (-100.0%)	Registers
7.1.7	SGPRs	48.00	112.0 (133.33%)	64.00	48.00	112.0 (133.33%)	48.00	112.0 (133.33%)	Registers

Fewer VGPRs
No AGPRs
More SGPRs

AMD together we advance_

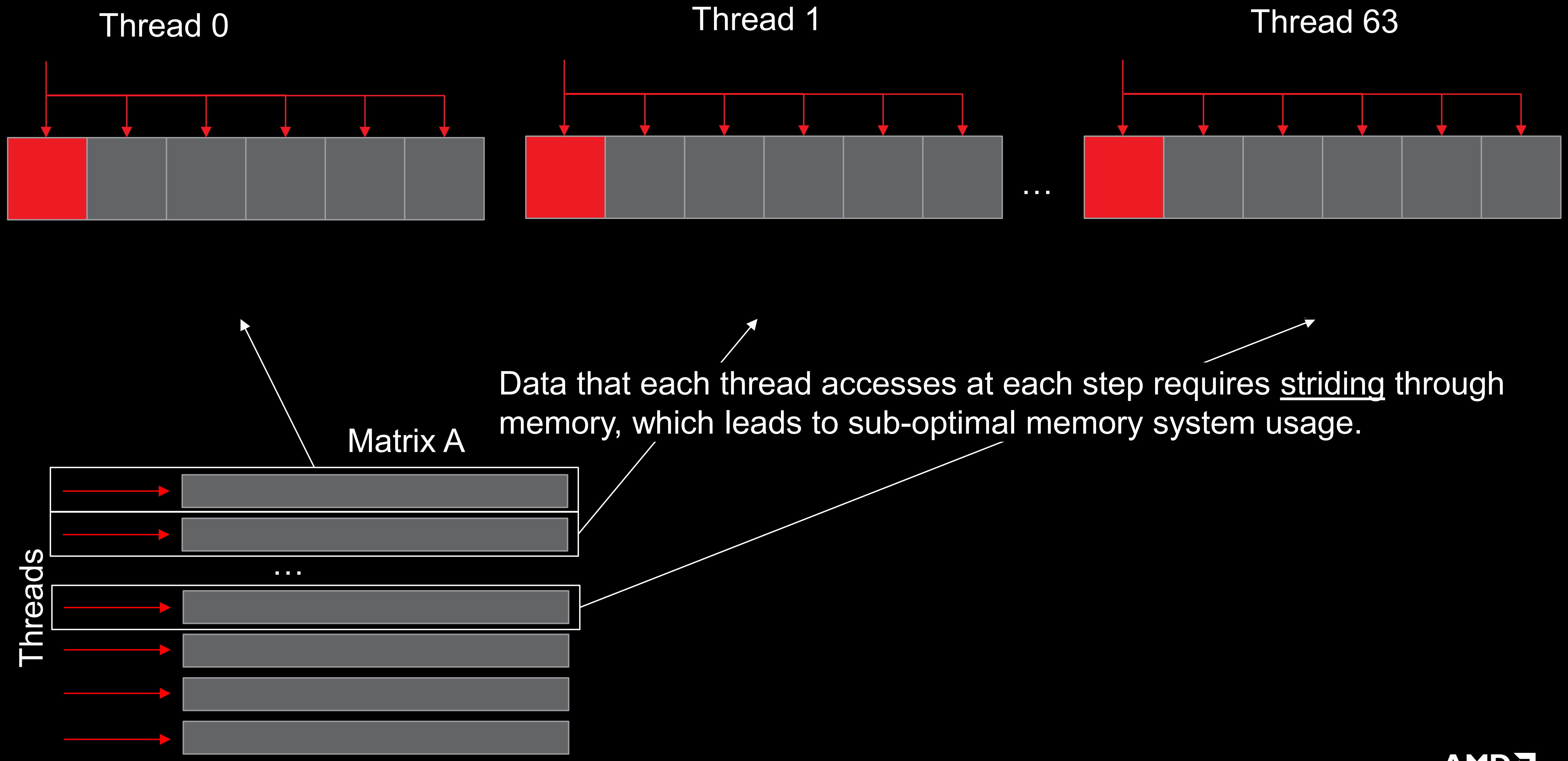
Exercise 3: Register occupancy limiter – takeaways

- In this case the occupancy limit is very minor
- Seemingly innocuous function calls inside kernels **can** lead to unexpected performance characteristics
  - Asserts, and even excessive use of math functions in kernels can degrade performance
  - Can be difficult to construct clear examples of this, anecdotally
- AGPR usage in the absence of MFMA instructions can indicate degraded performance
  - Spilling registers to AGPRs, due to running out of VGPRs
- To determine if any Workgroup Manager “insufficient resource” stats are nonzero, you can do:
  - `rocprof-compute analyze -p workloads/problem/MI300A_A1 --block 6.2`
    - Note: This will report more than just all “insufficient resource” fields

Exercise 4: Data access patterns important for performance

- The way in which threads access memory has a big impact on performance
  - If you increase occupancy and performance decreases, memory access patterns could be the culprit
- “Striding” in global memory has adverse effects on kernel performance, especially on GPUs
  - “Strided data access patterns” lead to poor utilization of cache memory systems
- These access patterns can be difficult to spot in the code
  - They are valid methods of indexing data
- May be less applicable to OpenMP® codes, but still useful to know what to look for
  - This example is more exaggerated than a reasonable code would be
- Using rocprof-compute can quickly show if a kernel’s data access is adversarial to the caches

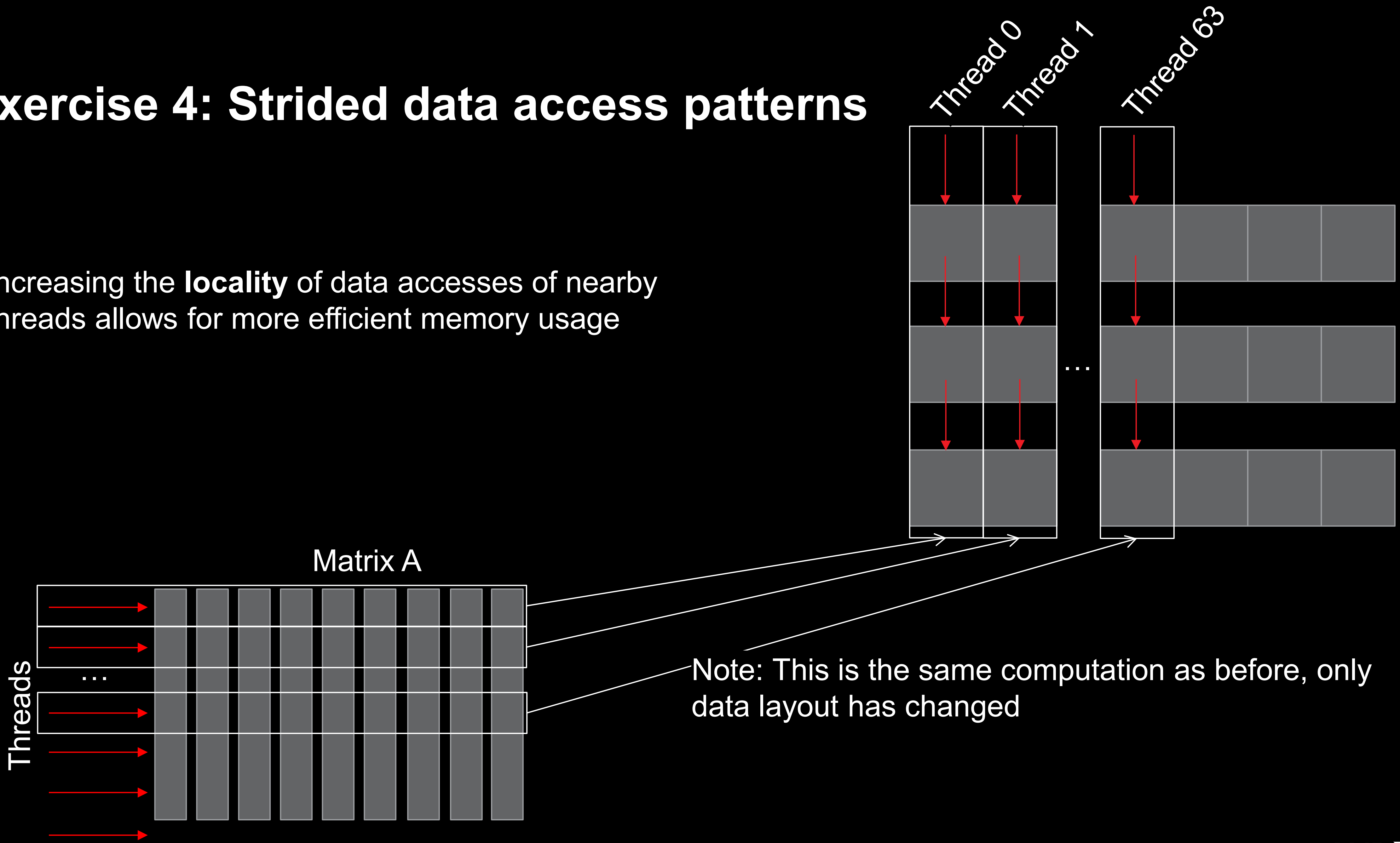
Exercise 4: What is a "strided data access pattern"?



Data that each thread accesses at each step requires striding through memory, which leads to sub-optimal memory system usage.

Exercise 4: Strided data access patterns

Increasing the **locality** of data accesses of nearby threads allows for more efficient memory usage



Note: This is the same computation as before, only data layout has changed

Exercise 4: Diagnose a strided data access pattern

- This exercise's setup makes it very easy to change the data access pattern
  - Generally, these optimizations can have nontrivial development overhead
  - Re-conceptualizing the data's structure can be difficult

- All the solution does is re-work the indexing scheme to better use caches
  - No required change to underlying data, because all the values in y, A, and x are set to 1

- Importantly, highly contended atomics on the same global memory address is bad coding practice. This code example does that, production codes should avoid this pattern (foreshadowing)

- To get started run:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`

Exercise 4: Strided data access pattern – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	100.00	100.0 (0.

~30% Slowdown?!

```
--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
16.1.0	Hit rate	0.01	75.0 (1061717.66%)	74.99	Pct of peak
16.1.1	Bandwidth	23.50	4.56 (-80.62%)	-18.95	Pct of peak
16.1.2	Utilization	85.08	96.69 (13.65%)	11.61	Pct of peak
16.1.3	Coalescing	25.00	25.0 (0.0%)	0.00	Pct of peak

+ ~75% in L1 hit

The solution better uses the L1, which should result in speedup. Why is the solution slower? Let's check atomic latency

```
--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
17.1.0	Utilization	98.80	98.57 (-0.23%)	-0.23	Pct
17.1.1	Bandwidth	55.85	2.73 (-95.12%)	-53.13	Pct
17.1.2	Hit Rate	93.66	0.68 (-99.28%)	-92.99	Pct
17.1.3	L2-Fabric Read BW	912.60	698.54 (-23.46%)	-214.07	Gb/s
17.1.4	L2-Fabric Write and Atomic BW	0.01	0.01 (-0.0%)	-0.00	Gb/s

L2 Cache Hit decreases sharply

Exercise 4: Atomic latency – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 17.2.11
```

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	100.00	100.0 (0.

```
0.2 Dispatch List
```

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	4

~30% Slowdown

```
--------------------------------------------------------------------------------
17. L2 Cache
17.2 L2 - Fabric Transactions
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
17.2.11	Atomic Latency	6289.38	10098.1 (60.56%)	3808.72	6289.38	10098.1 (60.56%)	6289.38	10098.1 (60.56%)	Cycles

Solution's atomic latency is higher!
This kernel is bound by atomics, not memory bandwidth

Exercise 4: Why is atomic latency higher in solution?

- In `solution.cpp`, we start hitting in the L1 cache, rather than having to go out to L2 for everything
- This reduces our memory latency, thus increasing the contention and pressure of the atomics
- This, coupled with the naïve, atomic-heavy reduction strategy, means atomics are our limiter, not cache

- This is the midpoint of the exercise, the lesson here is **not**: "use suboptimal cache access patterns"

- Let's try to optimize our reduction strategy to use a "shuffle reduction" to reduce the atomic contention
  - You can see how this is accomplished in `mi300a_problem` and `mi300a_solution`

- Note: In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere, and reduction strategies can be costly to refactor

Exercise 4: Atomic latency – relevant output

```
rocprof-compute analyze -p workloads/mi300a_problem/MI300A_A1 -p workloads/mi300a_solution/MI300A_A1 --dispatch 1 -block 17.2.11

INFO Analysis mode = cli
INFO [analysis] deriving rocprofiler-compute metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9593149.00	12351549.0 (28.75%)	9593149.00	12351549.0 (28.75%)	9593149.00	12351549.0 (28.75%)	100.00	100.0 (0.

```
0.2 Dispatch List
```

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	4

~30% Slowdown, still?

```
--------------------------------------------------------------------------------
17. L2 Cache
17.2 L2 - Fabric Transactions
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
17.2.11	Atomic Latency	6785.81	9603.13 (41.52%)	2817.32	6785.81	9603.13 (41.52%)	6785.81	9603.13 (41.52%)	Cycles

Exact values might be slightly different,
but conclusion stay the same

Solution's atomic latency is better, but
still much higher!

Exercise 4: Why is atomic latency *still* higher in solution?

- We already saw that solution uses the caches better, but this results in being bottlenecked by atomics
- We've seen that reducing atomic contention a small amount does not solve this, why?

- The atomic reduction is the bottleneck, and solution will always be slightly more contended than problem

- As our problem size grows, cache access and data movement should be our bottleneck

- This is the true lesson of this exercise: Profile a representative problem size!
  - Profiling problems that are too small may give you misleading optimization ideas

- Let's run `mi300a_problem` and `mi300a_solution` with larger problem sizes:
  - `rocprof-compute profile -n mi300a_problem_15 --no-roof -- ./mi300a_problem 15`
  - `rocprof-compute profile -n mi300a_solution_15 --no-roof -- ./mi300a_solution 15`

Exercise 4: Larger Problem Size – Relevant Output

```
rocprof-compute analyze -p workloads/mi300a_problem_15/MI300A_A1 -p workloads/mi300a_solution_15/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pc
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	309917571.00	25600803.0 (-91.74%)	309917571.00	25600803.0 (-91.74%)	309917571.00	25600803.0 (-91.74%)	100.00	10

~12x speedup

```
--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
16.1.0	Hit rate	0.00	75.0 (26214512.5%)	75.00	Pct of peak
16.1.1	Bandwidth	2.89	8.76 (202.8%)	5.87	Pct of peak
16.1.2	Utilization	81.82	98.35 (20.2%)	16.53	Pct of peak
16.1.3	Coalescing	25.00	25.0 (0.0%)	0.00	Pct of peak

Similar L1 performance to the small problem size.

We finally see the speedup we expect when using a better data access pattern

```
--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
17.1.0	Utilization	69.02	99.52 (44.19%)	30.50	Pct
17.1.1	Bandwidth	6.88	5.22 (-24.14%)	-1.66	Pct
17.1.2	Hit Rate	89.30	0.32 (-99.64%)	-88.98	Pct
17.1.3	L2-Fabric Read BW	173.83	1342.9 (672.53%)	1169.07	Gb/s
17.1.4	L2-Fabric Write and Atomic BW	0.01	0.0 (-0.0%)	-0.00	Gb/s

L2 hit rate is greatly reduced, and L2 bandwidth is greatly increased

AMD together we advance_

Exercise 4: Speed-of-Light cache access statistics

- The command below will show high-level details about L1 and L2 cache accesses:
  - `rocprof-compute analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 16.1 17.1`
- Ensuring better data locality will generally provide better performance
- In this case, we start hitting in the L1 cache, rather than having to go out to L2 for everything
- If you increase your cache efficiency but are running a small problem, you can check atomic latency:
  - `rocprof-compute analyze –p workloads/problem/MI300A_A1 --dispatch 1 --block 17.2.11`
- Note: In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere

rocprof-compute tips

- Filtering by kernel name and metrics during `rocprof-compute profile` will cut down on profiling time
  - `rocprof-compute profile -k “<kernel1>” “<kernel2>”` filters two kernel names
  - Surrounding kernel name in quotes allows spaces to appear in your kernel search string
  - Rocprof-compute applies wildcard automatically, so only unique kernel names substring required

- Use a subset of metrics for `rocprof-compute profile` to reduce the number of `rocprof` runs
  - `rocprof-compute profile --block SQ SQC -n <workload name> -- ./benchmark.sh`
  - `rocprof-compute profile --help` displays all block strings you can filter by
  - Performance model doc goes over some of the meaning behind lower-level hardware units and metrics

- rocprof-compute requires multiple app runs to collect hardware counters
  - Running with MPI is currently not supported, but this is being explored

- Don’t know where to start? → Easy things to check:
  - Are all the CUs being used? → If not, more parallelism is required (for most of the cases)
  - Are all the VGPRs being spilled? → Try smaller workgroup sizes
  - Is the code Integer limited? → Try reducing the integer ops, usually in the index calculation

Summary

- rocprof-compute: a GPU kernel-level profiling tool that automatically collects many counters
- Can create roofline analysis to understand kernel efficiency and distance to the theoretical peaks
- Displays many kernel metrics, but to correctly interpret it good knowledge of the kernel required
  - Easy to start running it, but steep learning curve for the analysis
- Supports standalone GUI, and CLI
- Includes several features such as:
  - System Speed-of-Light Panel
  - Memory Chart Analysis Panel
  - Vector L1D Cache Panel
  - Shader Processing Input (SPI) Panel

Hands-on exercises

- Located in our HPC Training Examples repo:

  https://github.com/amd/HPCTrainingExamples

- A table of contents for the READMEs if available at the top-level **README** in the repo

- Relevant exercises for this presentation located in:
  - rocprof-compute directory
  - Omniperf-OpenMP directory

- Instructions on how to run the rocprof-compute tests located in the specific example directories

- Log into the AAC node and clone the repo:

```
ssh <username>@aac6.amd.com –p 7000 -i <path_to_ssh_key>
git clone https://github.com/amd/HPCTrainingExamples.git
module load rocm/6.4.0 rocprofiler-compute/6.4.0
```

DISCLAIMERS AND ATTRIBUTIONS

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

AMD, the AMD Arrow logo, Radeon™, Instinct™, EPYC, Infinity Fabric, ROCm™, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board